SWDE - Software Development
Zusammenfassung FS 2019

Maurin D. Thalmann

18. Juni 2019

Inhaltsverzeichnis

1 Buildautomatisation

1.1 Sie kennen die Vorteile eines automatisierten Buildprozesses

- Automatisierter Ablauf, keine Interaktion mehr benötigt
- Reproduzierbare Ergebnisse
- lange Builds können auch über Nacht laufen
- Unabhängig von Entwicklungsumgebung

1.2 Sie können verschiedene Beispiele von Buildwerkzeugen benennen

Make (für C/C++ Projekte), Urvater der Build Tools,
hohe Flexibilität, gewöhnungsbedürftige Syntax
Ant Java mit XML
Maven Java mit XML
Buildr Ruby-Script
Gradle Groovy Script mit DSL
Bazel Java mit Python-like Scripts

1.3 Sie beherrschen die Anwendung eines ausgewählten Buildwerkzeuges (Apache Maven)

Beherrschen muss man es selber, es kann entweder aus der Shell (Terminal/Konsole) verwendet werden oder aus den integrierten Funktionen in der IDE selbst.

1.4 Sie sind mit den wesentlichen Konzepten von Apache Maven vertraut

Deklaration des Projektes in XML, zentrales Element pro Projekt ist das **Project Object Model (POM)**, welches Metainformationen, Plugins und Dependencies definiert. Basiert auf einem globalen, binären Repository. Plugins werden durch Dependencies dynamisch ins lokale Repository geladen ($HOME/.m2/repository)
Bei einem Buildprozess durchläuft ein Projekt einen Lifecycle mit folgenden Phasen:

validate validiert Projektdefinition
compile Kompiliert die Quellen
test Ausführen der Unit-Tests
package Packen der Distribution
verify Ausführen der Integrations-Tests
install Deployment im lokalen Repository
deploy Deployment im zentralen Repository

2 Modularisierung - Module, Komponenten, Schnittstellen

2.1 Sie wissen, was unter dem Begriff Modularisierung zu verstehen ist

«Ein grosses Ganzes in mehrere, sich abgeschlossene Einheiten (Module) aufteilen»
Flexible Zusammenstellung, Durchführung und Prüfung der einzelnen Module. Zwischen den Modulen können aber auch Abhängigkeiten bestehen.

2.2 Sie kennen die Begriffe Modul, Library, Komponenten und Schnittstelle auf der Ebene des Softwaredesigns

Kleinste Einheit: Klasse (Methoden/Daten/Attribute) $\leftrightarrow$ **Grösste Einheit:** vollständiges Softwaresystem

Modul In sich abgeschlossene und austauschbare Einheiten,
soll nur über seine Schnittstellen verwendet werden können $\rightarrow$ lose Kopplung;
Starke Kohäsion (möglichst in sich abgeschlossene Aufgabe erfüllen) $\rightarrow$ Information Hiding

Schnittstelle lässt Module untereinander interagieren / austauschen

Komponente strengere Form eines Moduls

Library Eine Sammlung thematisch zusammengehörender Funktionen (z.Bsp. Kalendermodul, Trigonometriemodul, etc.)

Die einzelnen Begriffe werden in einem späteren Lernziel ausführlicher beschrieben.

2.3 Sie können die Begriffe auf verschiedenen Abstraktionsebenen in einen sinnvollen Zusammenhang und Kontext setzen

Kopplung Ausmass der Kommunikation zwischen Modulen (Abhängigkeit zw. Modulen)

Kohäsion Ausmass der Kommunikation innerhalb eines Moduls (interner Zusammenhalt)

Ziel $\rightarrow$ Maximierung der Kohäsion, Minimierung der Kopplung

Gruppierung Modulen mit gemeinsamen Eigenschaften als Gruppe handhaben.
Beispiel: Modul für Datenexport in versch. Formate

Hierarchie (Rekursiv) Modul fasst mehrere (Sub-) Module zu einem zusammen.
Beispiel: Persistenzmodul als Datenspeicher mehrerer Entitäten

Geschichtet Modul(-gruppen) können logische Kette bilden, die vertikal als Schichten abgebildet werden.
Beispiel: Schichtenarchitektur, OSI-Referenzmodell, etc.

2.4 Sie sind in der Lage ein System zu analysieren und darin sinnvolle Module zu identifizieren

2.4.1 Modul

Kriterien für Entwurf von Modulen:
Zerlegbarkeit / Dekomposition
möglichst unabhängig voneinander, können einzeln genutzt/wiederverwendet werden
Kombinierbarkeit
sollen in anderem Umfeld wieder einsetzbar sein
(Zerlegbar, um auf andere Art wieder kombiniert werden zu können)
Verständlichkeit
unabhängig und in sich abgeschlossen verständlich sein, kann aber trotzdem hohe Komplexität erreichen.
Stetigkeit / Stabilität / Kontinuität
Struktur soll sich nicht stetig verändern, Aufteilung soll robust gegenüber Änderungen sein. Änderungen sollen sich auf eine minimale Anzahl Module beschränken.

Arten von Modulen:
Bibliothek/Library: beschrieben unter Kapitel 2.2
Abstrakte (komplexe) Datentypen: Modul implementiert neuen komplexen Datentyp und stellt darauf definierte Operationen zur Verfügung (Bsp. Komplexe Zahlen, Koordinatendarstellung, etc.)
Modellierung/Abstraktion physischer Modelle: Modul abstrahiert reales, physisch existierendes System (z.Bsp. Sensor, Gerätetreiber, Anzeigemodul, etc.)
Modellierung/Abstraktion logisch-konzeptioneller Systeme: Modul abstrahiert ein nur rein logisch existierendes System und macht es für eine höhere Abstraktionsebene nutzbar (z.Bsp. Grafikmodule, Datenbankmodule, Messaging, GUI-Module, etc.)

Definition eines Moduls:
Verhalten: Funktionalität des Moduls?
Export: Was bieten wir, Schnittstelle, um das Verhalten des Moduls für andere Module verfügbar zu machen.
Import: Was brauchen wir, von welchen Schnittstellen ist das Modul evtl. abhängig? (Dependencies)

Herausforderung bei Modulen:
Basiskonzepte: Hohe Kohäsion, lose Kopplung, starke Datenkapselung & Information Hiding
Vier Kriterien: Zerlegbarkeit, Kombinierbarkeit, Verständlichkeit & Stetigkeit
Verschiedene Arten: Bibliotheken, abstrakte Datentypen, physische / logische Modelle, Komponenten, etc.

2.4.2 Komponente

> *Eine Softwarekomponente ist ein Softwareelement, das zu einem bestimmten **Komponentenmodell** passt und entsprechend einem Composition Standard ohne Änderungen mit anderen Komponenten verknüpft und ausgeführt werden kann.*

Eine **Komponente** erfüllt strengere Kriterien als ein Modul und benötigt meist einen **Kontext**. Komponenten bedienen sich spezifischer Laufzeitumgebungen (z.Bsp. Container) in welche die Komponenten integriert (installiert, deployed, etc.) werden und dort lauffähig sind (Container stellen den Komponenten Basisdienste z.Bsp. für Lifecycle und Kommunikation bereit)
Komponenten können teilweise **dynamisch zur Laufzeit** ergänzt/entfernt/ausgetauscht werden
→ «Hot-Deployment» / Plugin-Mechanismen
Komponenten bieten Funktionalität an und sind von Funktionalität des eingesetzten Komponentenmodells (Framework/Produkt) **abhängig**.

Komponentenmodelle: konkrete Ausprägungen des Paradigmas der komponentenbasierten Entwicklung in Form eines Standards, Frameworks, Produktes. Schnittstellen für Interaktion und Komposition von Komponenten festlegen: wie kommunizieren Komponenten untereinander/mit dem Container Idealerweise: Komponentenmodell unabhängig von Gremium standardisiert (kann somit in unterschiedlichen Ausprägungen von versch. Herstellern implementiert/genutzt werden)

> **Beispiele**: Microsoft DCOM/ActiveX/.NET Remoting Services (WCF), CORBA (Common Object Request Broker Architecture), Enterprise Java Beans, OSGi (Open Services Gateway initiative / Alliance)

Komponente wird wie Modul über *Verhalten, Export, Import* definiert.
Zusätzlich wird ein **Kontext** verlangt: definiert notwendige Rahmenbedingungen, die für Betrieb der Komponente notwendig sind.

2.4.3 Schnittstelle

Schnittstellen werden konsequent für die kontrollierte Kommunikation zwischen Modulen oder Komponenten verwendet. **Vorteile** davon:

- Schnittstelle ist einfach verständlich, einfacher als die Implementierung.
- Schnittstellen helfen Abhängigkeiten zu reduzieren, vermeiden Abhängigkeiten zur Implementierung.
- Schnittstellen erleichtern Wiederverwendung.

Beziehungen zwischen einzelnen Teilen einer Software werden über Schnittstellen realisiert. Module konzentrieren sich auf ihre lokalen Probleme, Architektur definiert und hält Fäden (Beziehungen) des Systems zusammen.
Schnittstellen sollen minimal und schmal sein $\rightarrow$ aussagekräftige Methoden, präzis typisierte Parameter,
Methoden sollen möglichst:

- keine Überschneidungen haben
- keine globalen Daten verwenden
- statuslos (stateless) sein

Service: abstrahierte Schnittstelle, definiert sich primär über Fachlichkeit, dahinterliegende Technik idealerweise vollständig isoliert (Bsp. Webservice, wird über Web-Protokolle angeboten und abstrahiert die Implementation [Plattform, Sprache, Technologie] vollständig)
API: *(Application Programming Interface)*, technisch orientierte Schnittstelle, welche die Anbindung einer Komponente auf Quellcodeebene definiert (Bsp. JDBC [Java Database Connectivity], einheitliche Schnittstelle zur Kommunikation mit versch. DBMS)
Ebene **Objektorientiertes Design**: Schnittstelle = Java Interface
Ebene **Modularisierung**: Schnittstelle = logische Zusammenfassung versch. Artefakte (Klassen, Interfaces, Konfigurationsdateien, Doku etc.)

2.5 Sie kennen verschiedene organisatorische und technische Varianten um eine sinnvolle Modularisierung in der Entwicklung und im Deployment einzusetzen

2.5.1 Java 8

Module/Komponenten mit Klassen und Interfaces realisiert, Deployment meist als JAR. Klassen können sich an «Java Bean Spezifikation» halten

→ Default-Konstruktor, Setter/Getter, PropertyChange, Serialisierbar, etc.

Schnittstellen mit Java-Interfaces (zu class-Dateien kompiliert)
Komplexere Schnittstellen: mehrere Interfaces in Package zusammenfassen
Java 1.8 unterstützt selber keine Modularisierung
Information Hiding (einzelne Elemente vor Zugriff schützen) durch Packages, Sichtbarkeiten und Import, ermöglicht Zusammenfassen von Klassen/Interfaces in Gruppen, aber keine explizite Möglichkeit, Exports & Dependencies zu deklarieren
manifest.mf enthält Infos zu Identifikation, Herkunft und Version
Schnittstellen in getrennten JAR's (Modulen) verteilen → einfacher Austausch unterschiedlicher Implementationen
Workaround: Namenskonventionen und hohe Disziplin

2.5.2 Java 9

Modularisierung möglich, drei Ziele im Vordergrund:

- **Reliable Configuration:**
  fehleranfälligen Classpath durch auf Modul-Abhängigkeiten basierenden Modul-Path ablösen
- **Strong Encapsulation:**
  Modul definiert explizit sein öffentliches API. Auf alle restlichen Klassen ist von aussen kein Zugriff mehr möglich (auch wenn public).
- **Scalable Platform:**
  Java-Plattform selber wurde modularisiert, so können für Anwendungen individuell angepasste, schlankere Runtime-Images gebaut werden.

Weiteres zu Modularisierung in Java 9:

- Java-Packages neu in Modulen zusammenfassbar (zusätzliche Strukturebene in der Dateiablage, eindeutige Namensgebung analog zu Packages)
- Pro Modul wird ein *module-info.java* definiert (explizite Definition von Imports/Exports/Abhängigkeiten)
- Start einer Applikation: Laufzeitprüfung wird ausgeführt, ob alle notwendigen Komponenten vorhanden sind.
- Ende der «JAR-Hell»: Neues Format *jmod*, Class-Path wird durch Modul-Path abgelöst
- Vollständig rückwärtskompatibel

3 Dependency Management

3.1 Sie haben ein grundsätzliches Verständnis von Dependency Management

Dependency Management Beschreibt die Organisation und Techniken für Umgang mit Abhängigkeiten mi anderen Modulen.
(„Modul" hier vereinfacht als Überbegriff für Package / Library / Bundle / Komponente verwendet)
Abhängigkeiten können auf **interne** (Modul aus demselben Projekt) oder **externe** (Dritt-Modul aus anderem Projekt / Organisation) Module bestehen. Abhängigkeiten werden typisch in binärer / kompilierter Form aufgelöst, wozu Binärrepositories und Packagemanager(-tools) eingesetz werden.
Allen Systemen / Repositories ist gemeinsam:

- Zentrale Ablage auf einem (oder mehrere) Server
- standartisiertes Format
- zusätzliche Metainformationen
- typisch mit Abhängigkeiten (Dependencies) versehen
- Sicherung der Konsistenz (bspw. über Hash-Mechanismen)
- geregelte Zugriffsprotokolle
- Suchmöglichkeiten u.v.m.

Beispiele von populären Systemen / Repositories für DM und PM:
NuGet Package Manager für .NET-Plattform
apt Advanced Packaging Tool - Packetverwaltung für Linux
Yum Yellowdog Updater, Modified - Packetverwaltung für Linux
P2 OSGi-basiertes Komponentensystem
npm Node Packet Manager für JavaScript / node.js
Gems Packetmanager Ruby

3.1.1 Maven Repository

- Verschiedene öffentliche Repos **(OSS)**: Maven Central, JFrog JCenter (BinTray)
- Keine Schreibrechte auf öffentliche Repos (nur ausgewählte Personen über definierte Prozesse)
- Professionelle Organisationen betreiben interne Repositories, meist als lokaler Speicher und Mirror von öffentlichen Repos, professionelle Produkte wären:
  Apache Archiva, JFrog/Artifactory, Sonatype/Nexus
- Alle heruntergeladenen Artefakte vom Repo werden in lokalem Repo auf dem Rechner gespeichert (Caching) unter `$HOME/.m2/repository`
- Zur Verwendung muss unter `$HOME/.m2` die `settings.xml` Datei angepasst werden, damit die Dependencies nicht vom öffentlichen Repo geholt werden

3.2 Sie wissen wie am Beispiel von Java und Apache Maven das Dependency Management funktioniert

- Binäre Module (kompilierte Projekte) werden unter Java typischerweise als JAR-Dateien ausgetauscht (oder EAR, WAR, RAR etc.)
- Java kennt selber kein Verfahren zur Defintion von Abhängigkeiten zwischen Modulen und deren zentraler Verwaltung
  (Ab Java 9: Modularisierung (Jigsaw), aber ohne Versionierung)
- Früher: JAR-Dateien von Hand in Projekte kopiert
  → fehleranfällig, hohe Redundanz, hoher Platzbedarf etc.
  Heute ist Buildsystem Maven sehr populär geworden
- Unterscheiden zwischen:
  - **Format** für die zentrale Ablage der meist binären Artefakten mit Metainformationen im Repository
  - **Werkzeug**, um Artefakte von Repos zu suchen, beziehen, deployen, ggf. auch verwalten
- Repositoryformat Maven mittlerweile Standard, jedoch grosse Vielfalt bei Werkzeugen:
  - Apache Ivy - einziges „reines“ DM-Tool
  - Apache Maven - in Buildtool integriert, Original
  - DM anderer Buildtools basiert ebenfalls auf Maven-Repos:
    Buildr, Groovy Grape, Gradle/Grails, SBT etc.

3.2.1 Maven - Identifikation & Dependencies

Maven Projekt identifiziert sich mit drei Attributen (**maven coordinates**)

- **GroupId**:
  Reverse Domain Name der Organisation mit Zusatz für OE bspw. Projektgruppe
  Bsp: `ch.hslu.swde`
- **ArtifactId**:
  Name des Projekts bzw. enthaltene Module
  Bsp: `vereinsverwaltung-service`
- **Version**:
  Empfohlen: dreistellige Versionsnummer (Semantic Versioning)
  Bsp: `4.0.1`

Diese werden im `pom.xml` des Maven Projekts deklariert, damit soll eine Dependency weltweit absolut eindeutig identifiziert werden können:

```
<groupId>ch.hslu.swde</groupId>
<artifactId>vereinverwaltung-api</artifactId>
<version>4.0.1</version>
```

Benötigte Dependencies werden im `pom.xml` unter `<dependencies>` als ein `<dependency>`-Element eingetragen. Diese werden beim Build automatisch vom Repo runtergeladen und im lokalen Repository (`$HOME/.m2/repository`) gespeichert. Der Buildprozess referenziert die Artefakte dort mit entsprechendem Classpath:

```
<dependency>
   <groupId>ch.hslu.swde</groupId>
   <artifactId>vereinverwaltung-api</artifactId>
   <version>4.0.0</version>
   <scope>compile</scope>
</dependency>
```

3.3 Sie sind mit den Begriffen «dependency scopes» und «transitive dependencies» vertraut und können diese erklären

3.3.1 Dependency Scopes

`<scope>` in einer Dependency qualifiziert den Zweck und Geltungsbereich einer Dependency (wird empfohlen). Abhängig von Scopes sorgt Maven für spezifische Classpaths, woraus sich eine *implizite Verifikation des Designs* ergibt. Maven kennt viele Scopes, die Wichtigsten sind:

- **compile**
  (Default) Dependency wird für Kompilation und Laufzeit des Programms benötigt
- **test**
  Dependency nur für Kompilation und Ausführung der Testfälle benötigt (JUnit, AssertJ, Mockito)
- **runtime**
  Dependency nur für Laufzeit, aber nicht für Kompilation,
  bspw. für dynamisch geladene Implementationen

3.3.2 Transitive Dependencies

Maven bietet Feature zur Auflösung von transitiven Dependencies.

Abbildung 1: Transitive Abhängigkeit zwischen 3 Modulen

Auflösung der Dependencies:

- **Modul A** ist von **Modul B** abhängig, dieses wiederum von **Modul C**
- **Modul A** ist also transitiv auch von **Modul C** abhängig
- Für Kompilation wird also **Model C** auch benötigt

- Durch direkte & transitive Abhängigkeiten können auch Konflikte oder Zyklen auftreten.
  Maven erkennt und meldet diese, einfachere Konflikte können automatisch aufgelöst werden.
- Maven wertet die Dependencies als Graph aus, dient der Suche von Zyklen und Auflösung von Konflikten z.Bsp. über den kürzesten Pfad.

3.4 Sie kennen das Versionskonzept und die Funktionsweise von Snapshots.

- Einsatz von *Semantic Versioning* wird empfohlen
- Einmal deployte Version kann im Optimalfall nicht mehr überschrieben werden
  $\rightarrow$ nachvollziehbare Buildprozesse

Semantic Versioning

- **Major**-Release (**X**.x.x)
  Veränderungen in API, fachlicher Funktion und/oder in Konfiguration, welche zu früheren Versionen nicht kompatibel sind. Meist sind Anpassungen notwendig.
- **Minor**-Release (x.**X**.x)
  Erweiterungen in API, fachlicher Funktion oder Konfiguration, ist aber rückwärtskompatibel. Ohne Nutzung der Neuerungen keine Anpassungen notwendig.
- **Bugfix**/**Maintenance**-Release (x.x.**X**)
  Reine Korrekturen in Änderungen oder Implementation, voll rückwärtskompatibel, keine neuen oder veränderten Funktionen. Direkter, sofortiger Einsatz möglich/notwendig (Bugfix)

Versionierung mit Snapshots

- Version kann das Appendix `-SNAPSHOT` tragen.
- Gilt dann als erneuerbar und noch nicht stabil, sondern in Entwicklung
- Wird bei jedem **Build** vom Repo aufgelöst und aktualisiert
- Im Repo sind Snapshots mit Timestamp versehen

3.4.1 Managed Dependencies in Multi-Modul-Projekten

- Mehrere Submodule können von gleicher Dependency abhängig sein
- In jedem Modul sollte dieselbe Version verwendet werden
- **Lösung:** Im Master-POM über `dependencyManagement`-Element eine Liste an Dependencies (inkl. Version und Scope) als Baseline / Valid Version Set vordefinieren
  → Submodule müssen nur noch GroupId und ArtifactId angeben. Version und Scope werden vom Parent-POM vererbt.
- ***Alternativ:*** Verwendung eines BOM (Bill of Material):
  Verschiedene Versionen werden in „virtueller" Release-Unit als „Baseline" referenziert. Lieferant bestimmt, welche zueinander passenden Versionen eingesetzt werden. Das BOM wird selber als Dependency definiert.

3.5 Sie wissen auf welche Art Dependencies deployed werden

- Häufigste Deployment-Art: JAR-Dateien
- Beispiel für ein Artefakt
  `ch.hslu.vsk:stringpersistor-api:4.0.1`:
  POM (Metainfos) `stringpersistor-api-4.0.1.pom`
  JAR (Binary) `stringpersistor-api-4.0.1.jar`
  JavaDoc `stringpersistor-api-4.0.1-javadoc.jar`
  Source (bei OSS) `stringpersistor-api-4.0.1-sources.jar`

- Deployment in öffentliche Repos: sehr restriktiv
  → nichts mehr ändern, nichts löschen: Stabilität von Builds wahren!
- **Lösung:** nachvollziehbarer, automatisierter, verifizierbarer Release-Prozess, welcher von einem Build-Server ausgeführt wird

4 Versionskontrollsysteme - Source Code Management (SCM) / Version Control Systems (VCS)

4.1 Sie kennen die Aufgaben eines Versionskontrollsystems und können grundlegend damit arbeiten

Grundlegende Arbeit:
checkout lokale Arbeitskopie eines Projekts erstellen
update Änderungen Dritter in Arbeitskopie aktualisieren
log Bearbeitungsgeschichte eines Artefakts ansehen
diff verschiedene Revisionen miteinander vergleichen
commit / **checkin** Artefakte ins Repository schreiben → aussagekräftiger Kommentar!
Tagging: Revisionsstand mit Namen markieren, Marke oder Version: 1.5.2beta o.ä. Nützlich bei Release eines Produkts (aber auch meilensteine, Testversionen, Auslieferungszustände, etc.) wird unterschiedlich realisiert.

Branching: Parallele, voneinander getrennte Entwicklungszweige (für Bugfixing, Prototypen, Tests, Experimente, nachvollziehbare Änderungsworkflows, etc.) Bei Nicht-Wegwerf-Entwicklungen später Merging möglich/notwendig.

Ausschliesslich Quell-Artefakte werden verwaltet, **NIE** generierte/erzeugt Artefakte einchecken, können mit Hilfe der SCM ignoriert werden (.gitignore)

4.2 Sie kennen die verschiedenen Konzepte und Arten von Versionskontrollsystemen

- Zentrale oder verteilte Systeme
- Optimistische und pessimistische Lockverfahren
- Versionierung auf Basis einer Datei, Verzeichnisstruktur oder der Änderung (changeset)
- Transaktionsunterstützung (vorhanden oder nicht)
- Verschiedene Zugriffsprotokolle und Sicherheitsmechanismen
- Integration in Webserver (vorhanden oder nicht)

4.3 Sie können mit verschiedenen (Client-)Werkzeugen von Versionskontrollsystemen alleine und im Team arbeiten

CVS UT-Versionskontrollsystem, stabil, wenig Fehler, einfache Anwendung, ABER nur dateibasierend, Verzeichnisstruktur nicht versioniert, unterscheidet zwischen Text- und Binärdateien, Ablage von Binärdateien platzintensiv, keine Transaktionen

Subversion Transaktionsorieniert, versioniert ganze Verzeichnisstruktur, optimierte/effiziente Speicherung und Übertragung, Repositorystruktur frei wählbar (für Experten flexibler, für Anfänger schwieriger), Integration in Webserver möglich, aber Branching und Tagging technisch eig. Kopien/Links

git verteiltes System, primär lokale Arbeit, beliebig viele Server/Repos möglich, auch rein lokal einsetzbar, skaliert, Integration mit zusätzlichen Web-Applikationen, erfordert aber ein solides Konzept, für Einsteiger schwierig, da sehr mächtig und viele Funktionen

5 Testing Grundlagen

5.1 Sie kennen die Motivation, den Sinn und den Zweck des Testens, was Sie mit Tests erreichen können und was nicht

> **Qualität** ist die Übereinstimmung mit den Anforderungen unter gleichzeitiger Einhaltung von Qualitätskriterien.

Anforderungen müssen überprüfbar formuliert sein, typisch in Form von System- und Testspezifikationen.
Qualitätskriterien sind: Funktionalität, Zweckdienlichkeit, Robustheit, Zuverlässigkeit, Sicherheit, Effizienz, Benutzbarkeit, Geschwindigkeit etc. Zur Überprüfung von diesen stehen uns Methodiken, Techniken, Vorgehensweisen etc. zur Verfügung.

- Wesentliche Tätigkeit ist das Testen - Qualitätssicherung durch Testen
- Wir überprüfen Verhalten eines Programms anhand der Spezifikationen
- Möglichst oft automatisiert teste, manuelles Testen ist aufwändig und zeitintensiv
- Begleitmassnahmen: Reviews, Entwicklungsprozess, Walkthrough, Metriken, Analysen, Regression, Automatisation etc.

5.1.1 Welche Fehler können wir finden?

- **Syntaktische Fehler:**
  Abweichungen von der Sprachsyntax (Schreibweise & Grammatik) → formale Fehler.
  Diese werden durch Compiler aufgedeckt!
  - Beispiel Sprache: *„Der Lesr list ein Buck“*
  - Beispiel Java: `if(i = 10) then i = 0;`
- **Semantische Fehler:**
  Abweichungen/Unstimmigkeiten, die auf Kenntnisse über Dinge beruhen → inhaltliche Fehler
  - Beispiel Sprache: *„Die Erdnuss ass einen Elefanten“*
  - Beispiel Java: `double kelvin= celsius-273.15d;`
- **Funktionale Fehler:**
  Abweichung von Anforderungen oder Erfüllung falscher/unerwünschter Anforderungen.
  Funktionale Fehler können **nur bedingt** durch Testen gefunden werden
  - Beispiel: *„Ein Liter Bier kostet ab jetzt 1000.-“*

Testen ist systematisches, gezieltes und möglichst effizientes „Durchprobieren“ nach verschiedenen Qualitätskriterien.
Systematisch, gezielt mit wohlüberlegten Eingabedaten / Testwerten
Effizient möglichst aussagekräftige Tests durchführen
Testen ist also **nicht** zielloses, chaotisches Pröbeln, sondern anspruchsvoller als man denkt.
Man kann nur Vorhandensein von Fehlern, aber nie die Abwesenheit von Fehlern beweisen. Nach Test weiss man nur, dass ein Programm für die getesteten Eingabedaten korrekt läuft.
→ Eingabedaten so wählen, dass auf möglichst viele Varianten von möglichen Eingabedaten rückgeschlossen werden kann. Also auch untypische Daten miteinbeziehen.
Beispiel bei einem Sortieralgorithmus:

- beliebige, zufällige Daten (**Normalfälle**)
- vorsortierte Daten, keine Daten (**Sonderfälle**)
- ungültige, nicht sortierbare Daten (**Ausnahmefälle**)

	Testfall	Eingabewert [°Celsius]	Erwartetes Resultat [Kelvin]
Normalfälle	1	20.0f	293.15f
	2	0.0f	273.15f
	3	-10.0f	263.15f
Sonderfälle (zulässig)	4	-273.15f	0.0f
	5	Float.MAX_VALUE – 273.15f	Float.MAX_VALUE
Ausnahmefälle (unzulässig)	6	-273.16f	unzulässige Temperatur
	7	Float.MAX_VALUE	Bereichsüberlauf

Abbildung 2: Tabelle für die Planung von Testfällen

5.2 Sie kennen verschiedene grundlegende Testarten und –verfahren

- **Unit Tests:**
  Nur eine Methode/Klasse/Einheit wird getestet
  Sehr klein, übersichtlich, einfach, schnell ausführbar, einfach automatisierbar (Ausführung & Validation)
- **Integrationstests:**
  Mehrere Klassen (Module/Teilsysteme) in ihrem Zusammenspiel werden getestet
  Deutlich aufwändiger, aber auch automatisierbar
- **Systemtest:**
  Ganzes System (viele Klassen/Einheiten) wird getestet
  Klassisches Testen, erst spät möglich, aufwändig, auch automatisierbar
- **Black**- und **White**-Box Testing:
  Testen **ohne** oder **mit** Kenntnis der Implementation

5.3 Sie können in Ihrer Entwicklungsumgebung einfache und gute Unit Tests, basierend auf dem JUnit-Framework, implementieren und anwenden

5.3.1 Unit Tests

- Testfälle programmiert, somit automatisiert, jederzeit schnell ausführbar und wiederholbar
- Sind „self-validating“: Automatische Verifizierung, ob Testfall erfolgreich ausgeführt wurde
- Getestete Einheiten: einzelne Methoden oder Klassen
  → Kleine, überschauliche Testfälle
- Zahlreiche Frameworks für Testing, sind gut in IDEs integriert
  → Für Java: JUnit, UnitNG etc.

Nutzen daraus:

- Zeitersparnis: einmal schreiben, n-mal ausführen
- Kein manuelles „Testen“ mehr
- `System.out.println()`-Orgien entfallen
- Jederzeit reproduzierbar
- Automatische Verifizierung der Ergebnisse mit Reporting
- Testfälle „dokumentieren“ erwartete Funktion/Resultat
- Freiheit, mit geringem Risiko Code zu verändern / umschreiben (Refactoring)

5.3.2 JUnit Test Framework

- Aktuelle Version ist JUnit 5
- JUnit wird heute für alle automatisierbaren Testfälle eingesetzt
  → *Nicht jeder JUnit-Testfall ist ein Unit-Test!*
- JUnit 5 hat viele Verbesserungen, ist aber nicht rückwärtskompatibel (OMG ein Major-Release!)

Ein Testfall entspricht dem „Build-Operate-Check“-Pattern:

1. **Erstellen** der Testobjekte bzw. -daten
2. **Manipulieren** der Testobjekte bzw. -daten
3. **Verifikation** der Ergebnisse (mit `assert*`-Methoden)

Auch bekannt als „Triple-A“-Pattern: **A**rrange, **A**ct, **A**ssert

```
@Test
void testGetQuadrantInOne() {
   final Point point = new Point(4,5);        ❶
   final int quadrant = point.getQuadrant(); ❷
   assertEquals(1, quadrant);                 ❸
}
```

Abbildung 3: Beispiel-Testfall

Self-Validating mit assert*-Methoden Einfacher Vergleich zwischen einem Soll- und Ist-Wert, Methoden der Klasse `org.junit.jupiter.api.Assertions`, für viele Daten überladen (elementar oder Klasse `Object`)

```
assertEquals(long expected, long actual)
```
Prüft, ob die zwei long-Werte gleich sind.

```
assertEquals(float expected, float actual, float delta)
```
Prüft, ob zwei float-Werte gleich sind (mit Toleranz `delta`).

```
assertEquals(Object expected, Object actual)
```
Prüft, ob die Objekte gleich sind.

```
assertTrue(boolean condition)
```
Prüft, ob eine beliebige boolesche Bedingung `true` ist.

Abbildung 4: Beispiele für assert*-Methoden

Annotations für Test-Methoden Zur Konfiguration von Tests stehen mit JUnit verschiedene Annotationen zur Verfügung:

Annotation	Beschreibung
`@Test` `void testMethod()`	Markiert eine Methode als JUnit-Testfall (Pflicht). Darum ist die Namenskonvention **`test*`** eigentlich nicht mehr notwendig (aber trotzdem empfohlen!) Achtung: **`org.junit.jupiter.api.Test`**
`@Test @Disabled(…)` `void testMethod()`	Markiert einen Testfalls als (temporär) deaktiviert. Viel besser als auskommentieren, weil er so explizit als "skipped" ausgewiesen wird. Optional: **`String`**-Parameter mit Begründung.

`@BeforeAll`
`@BeforeEach`
`@AfterEach`
`@AfterAll`
`… void fooBar()`
	Markieren eine (bei `*All` statische) Hilfsmethode zur automatischen Ausführung vor/nach jedem/allen Testfällen in der Klasse.
	Optional, bei Unit Tests möglichst vermeiden.

Abbildung 5: Annotationen im Code für Testfälle

Namenskonventionen

- **Testklasse:**
  Für Klasse `Demo`: `DemoTest`
  `Test` als Appendix, alle Testklassen ind `/src/main/test`!
- **Testmethoden:**
  Für Methode `foo(...)`: `testFoo[Xyz]()`
  `test` als Präfix, `Xyz` als freie, ergänzbare Fallbeschreibung, Testmethoden haben keine formalen Parameter

5.3.3 Empfehlungen

- Besser viele kleine Testmethoden als wenig grosse
  → im Fehlerfall bessere Selektivität
- Wenige `assert*`-Statements pro Testmethode
  → höhere Selektivität im Fehlerfall und übersichtlicher
- Methoden mit Rückgabewert am einfachsten testbar
  → Bei void-Methoden ggf. indirekt über Statusabfragen auf getestetes Objekt testen
- Getter & Setter: gemeinsam oder indirekt „beiläufig" testen
- **Nie** Schnittstellen oder Sichtbarkeit nur für Testbarkeit ändern
- Klasse/Methode zu schwerig zu testen? → Hinterfrage mal deren Design...

5.4 Sie kennen die Vorteile von Test First

Wir testen **kontinuierlich** während der Implementation, um von Beginn an Gewissheit zu haben, dass es funktioniert.
Fehler finden, bevor man sie macht. Fehler finden, bevor man sie implementiert hat. Oder Fehler im Ansatz finden, bevor es jemand anders tut. Bullshitsätz...

5.4.1 Test First

Immer vor der Implementation die Testfälle schreiben!
Vorteile dabei:

- Beim Schreiben der Testfälle denkt man automatisch an Implementation des zu testenden Codes
  → Implementation „reift“ sozusagen beim Schreiben der Testfälle heran
- Ausnahmen und Sonderfälle fallen auf, welche bei der eigentlichen Implementation dann „automatisch“ berücksichtigt werden
- Sobald die Komponente fertig implementiert ist, kann sie sofort getestet werden

5.5 Code Coverage (Codeabdeckung)

- Code Coverage = Metrik, welche zur Lauftzeit misst, welche Quellcodezeilen von Testfällen abgearbeitet wurden
- Wird typischerweise bei Ausführung der Unit-Testfällen (bspw. durch JUnit) durchgeführt
  → oder zur „normalen“ Laufzeit, zur Messung welche Funktionen effektiv genutzt werden
- Ermöglicht eine Aussage, wie umfassend der Code getestet wurde
  (gezielte Effizienzsteigerung der Testfälle)
- Hohe Code Coverage ist **kein** Beweis für gute Testfälle oder Fehlerfreiheit!

5.6 Unit Tests - Bilanz

Positiv:

- Testen ist vollständig in Implementationsphase integriert → Aufgabe des Entwicklers
- Neue / veränderte Methoden sind unmittelbar, reproduzierbar & schnell testbar (regressiv)
- Test First: problemlos möglich und motivierend
- Automatisiertes, übersichtliches Feedback / Reporting
- Messung von Codeabdeckung integrierbar

Negativ:

- Qualität der Testfälle im Auge behalten → **Qualität vor Quantität**
- Für GUI(-Komponenten) aufwändiger
- In manchen Architekturen / Umgebungen schwierig umsetzbar

6 Automatisiertes Testing

6.1 Sie kennen die verschiedenen Testarten und sind in der Lage gute Unit-und Integrationstests zu schreiben

6.1.1 repeat(Unit Tests)

- Häufig mit Komponenten/Modul/Entwickler-Tests gleichgesetzt
- Funktionale Tests von einzelnen, in sich abgeschlossenen Einheiten
  (typisch Klasse, aber auch Komponente oder Modul)
- Ziele von guten Unit-Tests:
  - schnell, einfach ausführbar, selbstvalidierend (assert*-Methoden), automatisiert
  - mögl. ohne Abhängigkeiten zu anderen Klassen/Komponenten/Modulen (lose Kopplung)
  - Während Entwicklung geschrieben und ausgeführt (in IDE oder durch CI)
- Gute Unterstützung durch Frameworks (JUnit, TestNG etc.)

6.1.2 Integrationstests (JUnit)

- **Namenskonvention:**
  `XyzIT` für Integrationstest (Klassenname + Appendix `IT`)
  Ablegen unter `/src/test/java`
- Unterscheidung ergibt sich „nur“ durch:
  Zeitpunkt der Ausführung, Laufzeit und Abhängigkeiten während der Ausführung
- Eigene Apache Maven Stage (`integration-test`) für Integrationstests
  Getrennte Plugins `surefire` und `failsafe` weisen Testresultate auch getrennt aus
  (Unit und Integration)

6.1.3 Unit- vs. Integrationstests

- **Unit-Tests** sind wirklich Unit-Tests, wenn sie auf beliebigem System und jederzeit lauffähig sind und (bei Java) auf unterschiedlichen Betriebssystemen laufen
- **Konsequenz:**
  Testfälle, die mit Dateisystem interagieren oder Sockets verwenden (auch wenn nur auf `localhost`), sind streng gesehen bereits Integrationstests
- Unit-Tests dürfen somit nie aufgrund von „Fremdeinflüssen“ fehlschlagen
  (bspw. falscher Pfad, Platz, Zugriffsrechte etc.)
- **Empfehlungen:**
  - Bewusst zwischen beiden Kategorien trennen
  - Im Team auf gemeinsame Philosophie der Trennung der Kategorien einigen
    (Kann auch mal projektspezifisch sein)
  - Je mehr als Unit Tests realisiert wird, desto besser
  - Jeder automatisierte Test hilft
  - Hilfsmittel wie Code Coverage oder Mocking nutzen

6.2 Sie nutzen Werkzeuge zur Messung der Codeabdeckung aktiv zu Verbesserung Ihres Codes und der Testfälle

Ja klar tu ich das!

6.3 Sie kennen das Prinzip der Dependency Injection

- Schlechte Testbarkeit durch zu stark gekoppelte Abhängigkeiten
  → Schlechtes Design
- High-Level-Klassen sollen nicht von Low-Level Klassen abhängig sein,
  sondern beide von Interfaces
- Interfaces sollen nicht von Details, sondern Details von Interfaces abhängig sein!
- Isolation von Klassen, Auflösung von Abhängigkeiten durch Dependency Injection
- Dependency Inversion Principle - DIP (aus S.O.L.I.D.)

6.3.1 Dependency Injection - Beispiel

Eine Klasse `PersonPersistor` soll implementiert werden, welche `Person`-Objekte als `String` serialisiert und in einer Datei speichert. Es existiert eine Implementation `StringPersistorFile`, welche wiederverwendet wird:

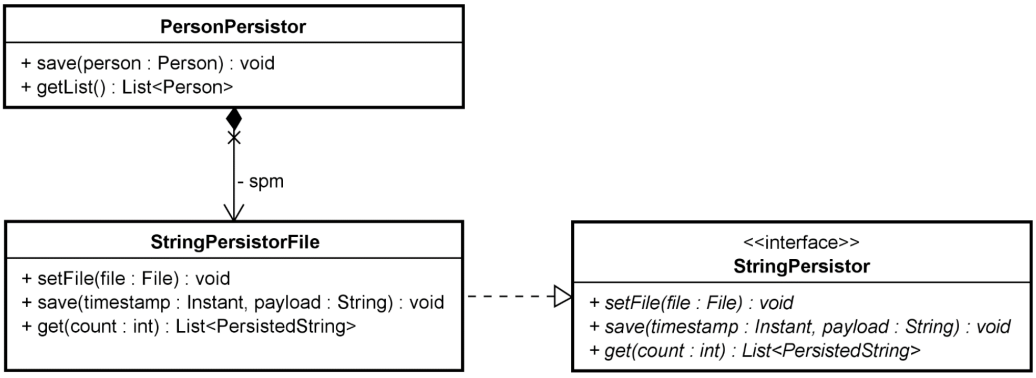


Abbildung 6: Erster Lösungsansatz für das Problem

EINE ZIEMLICH BEKNACKTE LÖSUNG FÜR TESTING!

```
public final class PersonPersistor {
   private final StringPersistorFile spm = newStringPersistorFile();
   ...
}
```

- Einfach für die Realisierung, ist aber der einzige Vorteil, negative Aspekte:
  - Typen und Implementation sind stark gekoppelt
  - Abhängigkeit zur Implementationsklasse (obwohl ein Interface existieren würde)
  - Unflexible Implementation
- Wir wollen die Funktionalität der `PersonPersistor` testen, um Strings zu serialisieren und wieder zu deserialisieren
- `StringPersistorFile` greift aber auf das Dateisystem zu, weshalb die Abhängigkeit uns in die Kategorie der **Integrationstests** wirft
  Ausserdem würden wir den bereits getesteten `StringPersistorFile` nochmals testen
  → Selektivität des Testfalls sinkt
- Man müsste die Abhängigkeit auf den intern verwendeten `StringPersistorFile` durch etwas anderes ersetzen können

Lösung auf der nachfolgenden Seite

Dependency Injection Eine Klasse/Komponente erzeugt ihre Abhängigkeiten nicht selber, sondern lässt sich diese (wahlweise) auch von Aussen übergeben.

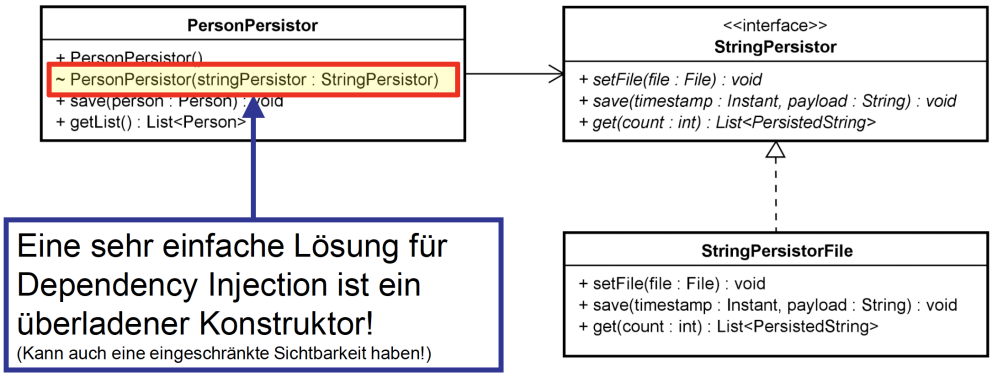


Abbildung 7: Besserer Lösungsansatz für Testing mit Dependency Injection

- Der konkrete Typ (Implementation) wird durch ein Interface ersetzt
  → Kopplung nimmt stark ab
- Es können so verschiedene, alternative Implementierungen genutzt werden
  → Andere Möglichkeiten / Ansätze testen
- Dependency Inversion Principle
  → Resultiert in besserer „Separation of Concerns“
- Testbarkeit wird massiv vereinfacht, man kann während Tests eine alternative Implementation als Platzhalter einfügen → **Test Double**
- Integrationstest werden wieder zu Unit Tests, schnellere & selektivere Tests

6.4 Sie wissen was Test Doubles sind und können Mocking-Frameworks einsetzen

6.4.1 Test Doubles

- „Double“ ist Platzhalter für eine echte, produktive Implementation während des Tests
- Oberbegriff „Test Doubles“ umfasst verschiedene, interessante Spezialisierungen
- Test Doubles sollen Aufwand für Integrationstests reduzieren, indem stattdessen mehr Testfälle als Unit-Tests realisiert werden
- Möglichst viel mit Unit-Tests prüfen, weil:
  - Erste Teststufe, direkt bei Entwickler
  - Schnell, häufig, überall lauffähig, vollständig automatisiert
  - Hohe Selektivität der Testfälle
- Test Doubles sind auch innerhalb von Integrationstests nützlich
  → Gezielte Isolation der Tests von einzelnen Integrationen (Abhäng. von anderen Systemen)
- Für Test Doubles muss gutes Design der Software vorliegen
  → Einsatz von Interfaces lohnt sich (fast) immer, da ein Interface verschiedene Implementationen zulässt, minimum eine „echte“ und eine Test-Implementation
- Wahl der Implementation muss zur Test-Laufzeit beeinflusst werden können
  → per Dependency Injection (manuell oder per Framework)
  (**Achtung:** Sicherung nötig, dass das nicht in der **Produktion** passiert)

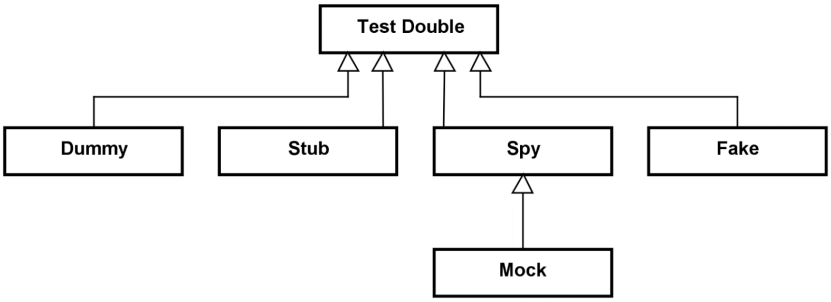


Abbildung 8: Übersicht über alle verfügbaren Test-Doubles

Dummy

- Sehr primitive, leere Ersatzimplementation, wird als aktueller Parameter an Methoden übergeben
  → Aktueller Parameter ist zwar notwendig für Test, dessen Nutzung und Implementation (für Test) aber irrelevant
- Dummy: dient funktionsloser Entkopplung der beim Test unerwünschten Abhängigkeiten

  Beispiel:
  Einem Objekt muss bspw. ein Logger übergeben werden. Dieser soll aber nichts machen, da Loggen nicht das eigentliche Testziel ist.

Stub

- Einfache Implementation, die mit möglichst geringem Aufwand sinnvolle, vordefinierte Werte (bspw. Konstanten)zurückliefert
- Erlaubt sogenanntes „State“-Testing
  → State durch Daten repräsentiert, State bei Stubs in der Regel konstant
- Für verschiedene Testziele ggf. auch mehrere untersch. Stubs erstellen

  Beispiel:
  Eine Klasse für Authentifizierung testen, welche...
  - Beliebige Nutzer/Passwörter akzeptiert: `login = true`
  - Niemanden akzeptiert: `login = false`

Spy

- Alternative Implementation, welche dynamische Werte zurückliefern kann. Der Spy merkt sich gleichzeitig auch die Aufrufe der Methode
  (Anzahl/Häufigkeit, Paramater, Zeitpunkt, Exceptions etc.)
- Nach Interaktion können aufgezeichnete Ereignisse zur Verifikation des Testfalls genutzt werden.
- Erlaubt „Behavior“-Testing (Verhalten)

  Beispiel:
  Wurde auf dem beim Testkandidaten registrierten `ActionListener`(-Spy)
  die Methode `actionPerformed()` auch tatsächlich aufgerufen?

Mock

- Spezialisierung des Spy, welcher dynamische Werte zurückliefern kann und die korrekte Interaktion selber verifizieren kann → Abgrenzung zum Spy
- Typisch mithilfe von Mock-Frameworks zur Laufzeit für jeden Testfall als individuelle Mock-Objekte erstellt (Proxy-Pattern, GoF)
  → Verhalten für jeden Testfall dynamisch / programmatisch konfiguriert
- Sehr ähnlich zu Spy, Unterschied ist jedoch der Ort der Verifikation, Mocks sind spezifischer

Fake

- Alternative Implementation, welche eine Komponente
  mit vernünftigem Aufwand vollständig ersetzen kann
- Ermöglicht vollständige Entkopplung von einer Abhängigkeit
  Trade-Off → Aufwand der Implementation des Fake muss in vernünftigem Verhältnis zum Nutzen sein (Unit vs. Integration)

  Beispiel:
  Abhängigkeit von Webservices durch lokale Fake-Implementation ersetzt
  - Kommunikation fällt weg, ist somit schneller
  - Implementation trotzdem vorhanden (wenn nicht zu komplex),
  idealerweise auch wiederverwendet

6.4.2 Effektives Testen mit Test Doubles - Beispiele

Mockito - bewährtes Testing-Framework für Java.
In Maven einbinden mit `org.mockito:mockito-core:2.23.0`
Viele statische Klassen auf der Klasse `org.mockito.Mockito`.
Bestenfalls statischer Import aller Funktionen:
`import static org.mockito.Mockito.*;`

Test mit Fake Für Testausführung wird Fake-Implementation `StringPersistorMemory` (statt `File`) verwendet, welche Strings nur im Memory „speichert“:

```
@Test
public void testGetEmptyList() {
   final Person Persistorinstance =
     new PersonPersistor(new StringPersistorMemory());
   assertThat(instance.getList()).isEmpty();
}
```

Für diesen Testfall resultiert:

- Keine Abhängigkeit mehr zum Dateisystem
- Unit-Test (kein Integrationstest)
- Testet viel weniger Dritt-Code und wird damit selektiver

Test mit Mock Für Testausführung wird Mock (oder Spy) verwendet, welcher direkt im Testfall erzeugt / konfiguriert wird:

```
@Test
public void testGetEmptyList() {
   finalStringPersistor mock =
                        mock(StringPersistor.class);
   when(mock.get(0)).thenReturn(Collections.emptyList());
   finalPersonPersistorinstance =
                        newPersonPersistor(mock);
   assertThat(instance.getList()).isEmpty();
}
```

Der Testfall:

- Keine Abhängigkeit mehr zu einer Implementation
- Selektivität ist maximal

6.4.3 Empfehlungen

- Gewisse Klassen sind zu aufwändig für Mocking
- Gewisse Klassen sind zu einfach für Mocking
- Verständlichkeit des Testcodes am wichtigsten, wenn Testfall durch Mocking zu kompliziert wird, dass man ihn nicht mehr versteht, hed mer huere verlore.
- Überleged si sich öbs sich lohnt. Stiiged si doch langsam id Technik ii, si esch absolut faszinierend! *- Roli Gisler*

Wann setzt man was ein?

- **Dummy** / **Stub:**
  Einfache Ersatzimplementationen, um bessere Testisolation zu erreichen
  Geringer Aufwand $\rightarrow$ höhere Selektivität / Stabilität der Testfälle (einfach)
- **Spy** / **Mock:**
  „Universalwaffen“ für Behavior-Testing mithilfe Mocking-Frameworks. Können auch zur Realisierung von Stubs / Dummies genutzt werden (komplexer)
- **Fake:**
  Aufwändige Implementation, zur vollständigen Entkopplung vom Original.
  Muss sich lohnen (aufwändig)
  (Ja aber was wänn de Fake besser isch as de Original..??)

- Design ist entscheidend:
  So oft wie möglich Interfaces verwenden, um schneller alternative Implementationen zu integrieren

7 Software Architektur

7.1 Sie können den Begriff «Software Architektur» einordnen

- Eine Architektur ist eine Abstraktion, etwas wird zusammenfassend/vereinfacht dargestellt
  → vergleiche Modellierung
- Architektur beschreibt ein System durch:
  - **Struktur und Aufbau:**
    Sub- & Teilsysteme, Schichtung, „Zwiebel“, Verteilung
  - **darin enthaltene Softwareteile:**
    Gegliedert nacht Aufgaben, Zuständigkeiten, Komponenten, aber auch Technologien
  - **deren Beziehungen untereinander:**
    Abhängigkeiten, Schnittstellen, Datenflüsse, Deployment

Definitionen zu Architektur

> *«Softwarearchitektur definiert sich durch die Kernelemente eines Systems, welche als Basis für alle weiteren Teile nur schwer und aufwendig verändert werden können.»*
> - Martin Fowler

> *«Die Architektur repräsentiert die signifikanten Designentscheidungen die ein System festhalten, wobei die Signifikanz an den Kosten von Änderungen bemessen wird.»*
> - Grady Booch

7.2 Sie haben eine Vorstellung, welche Informationen und Artefakte eine Software Architektur beschreiben

7.2.1 Übersicht über Aspekte der Softwarearchitektur

- Grundlegende Struktur
  - Schichten, Client / Server, n-tiers etc.
  - Architekturmuster
- Kommunikation & Verarbeitung
  - Kommunikationsmuster (bspw. synchron / asynchron)
  - Verteilbarkeit, Parallelität, Performance, Robustheit, Resilienz
- Eingesetzte Technologien
  - Userinterface (Fat-, Rich- oder Thin-Client)
  - Persistenz der Daten
  - Referenzarchitekturen
- Qualitätsaspekte
  - Wartungsfreundlichkeit, Erweiterbarkeit etc.

7.2.2 Motivation für (gute) Architektur

- Explizite Architektur macht ein System erst „verstehbar“. Fundamentale Grundlage für die Diskussion, Planung, Implementation und Betrieb eines Softwaresystems.
- Unmittelbare Qualitätsaspekte:
  - bessere Wartbarkeit (Bugfixing)
  - höhere Wiederverwendbarkeit
  - leichte Erweiterbarkeit
  - grössere Sicherheit
  - vereinfachte Testbarkeit
  - höhere Stabilität & Robustheit
- Auswirkung dieses Qualitätsaspekte:
  Verkürzung der time-to-market, echte Agilität

7.2.3 Architekturmuster und -stile

- Konzepte, beschreiben Grundaufbau eines ganzen Systems
  (Vergleich: Entwurfsmuster sind (objektorientierte) Konzepte für einzelne Teilprobleme)
- Architekturmuster für verschiedene zentrale Aspekte/Schwierigkeiten in Softwarearchitektur:
  für (grosse, komplexe), (stark) verteilte, inter(re)aktive etc. Systeme, diese zu strukturieren
  Beispiele: Client/Server, peer-to-peer etc.
- Typisch nicht so stark vereinheitlicht & standardisiert wie Entwurfsmuster
- Sehr viele Architekturstile hab sich im Laufe der Zeit entwickelt. Je nach Applikationsart erweisen sich diese als sehr schlecht oder gut brauchbar.
- **Erkenntnis 1:**
  Führte oft zum „FAT-Client“ (Geschäftslogik im Client) und einer überforderten Datenbank
  Vorteil: zentrale Datenhaltung, hohe Konsistenz (RI, Trigger, SP)
  Nachteil: Transaktionen, Locking, Ressourcen, Verteilung etc.
- **Erkenntnis 2:**
  „Im Kleinen“ bspw. für lokale (Einbenutzer-)Apps auf Handys funktioniert das gut.
- **Wahre Kunst:** Richtige Applikation am richtigen Ort!

Fliessender Wechsel zw. Design und Architektur Grosse Palette an „altbekannten“ Mustern, Prinzipien & Techniken: Modularisierung von Komponenten, Schnittstellen, Packages; Gruppierung dieser zu (Sub-)Systemen; Nutzung versch. Muster zur Strukturierung (bspw. MVC - Model View Control, Schichtenbildung etc.). **Herausforderung:** All diese Prinzipien auf den unterschiedlichen Abstraktionsebenen angemessen zu befolgen und zu beurteilen. Schlussendlich haben wir «nur» einen Haufen von Klassen und Schnittstellen - jede weitere Strukturierung ergibt sich weitgehend nur durch Organisation und Konventionen!

7.3 Versch. Arten von Applikationen

- Einzelbenutzerapplikation:
  Eher klein, evtl. nur Mobile App, lokale Persistenz, wenig komplexe Funktionalität
- Mehrbenutzerapplikation:
  bspw. Unternehmensapplikation, zentrale Services & Daten, beliebige Komplexität, typisch in mehrere (Teil-)Systeme aufgebrochen.
- Internetanwendungen:
  Typisch mit Web-GUI, beliebig viele Benutzer, verteilte Datenspeicherung, beliebige Komplexität

- Sehr unterschiedliche Anforderungen an Architektur!

7.4 Sie verstehen, dass die Bildung von Schichten eine fundamentale Basis für die meisten Architekturen ist

- **Schichten (Layers):** Gliederung eines Systems
  in aufeinander aufbauende, funktional getrennte Schichten
  - Kommunikation zw. Schichten über Schnittstellen
  - Abhängigkeit nur in Richtung der tieferliegenden Schicht
- Zur Strukturierung, innerhalb eines Systems & systemübergreifend
- Mögliche physische Verteilung der Schichten: Schichten $\rightarrow$ Tiers

7.4.1 Schichtenbildung

- Bildung nach versch. Kriterien:
  Logisch / Funktional, Technik, Abstraktionsebene etc.
- Kann hierarchisch verfeinert werden:
  1. funktionale Schichtung
  2. Abstraktionslevel
  3. Technologie
- Implementation mit Java:
  Schichten manifestieren sich in Modul- und Packagestruktur:
  `ch.domain.system.server` & ch.domain.system.server
  (Kann schlechter Ansatz sein, vergleiche „package by feature“)

Schnittstelle zw. Schichten

- Explizite Schnittstellen zw. Schichten → siehe *Modularisierung*
- Erlauben Entkopplung von der konkreten Implementation, die ausgetauscht werden kann
  → auch die vollständige Schicht
- **Achtung:** zwischen Schichten ausgetauschte Daten sind Bestandteil der Schnittstelle und tragen zur impliziten Kopplung bei
  - nützt nichts, wenn wir identische Datenobjekte über alle Schichten hinweg jagen
  - Entscheid nötig, bei welchen Schichten wir Schnittstellen auch über Datentypen brechen

Verteilte Schichten auf Tiers

- Schichtenbildung:
  Fundamentale Grundlage zur Realisierung von verteilten Anwendungen / Architekturen
- Schichtengrenzen:
  Eignen sich gut zur Auftrennung, um Teile auf verschiedene Systeme zu deployen / verteilen
- Werden Schichten auch physisch getrennt, spricht man von Tiers
  (Client / Server-Beziehung zwischen Schichten)

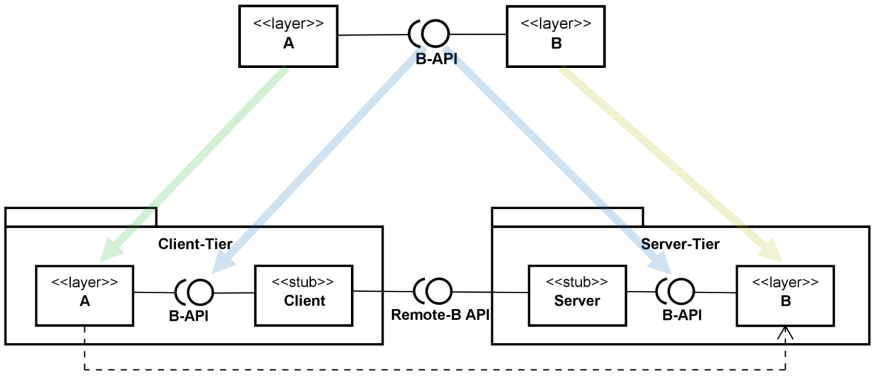


Abbildung 9: Aufteilung von Schichten auf Client / Server - Tiers

7.4.2 Logische 3/6-Schicht-Architektur

Aufteilung in drei fundamentale Schichten:

- **Präsentation** (Presentation / [GU]I Layer):
  Visualisierung, User-Interface, Logik
  (bspw. Bestellmaske für Artikel)
- **Geschäftslogik** (Business[logic] Layer, Domain Model):
  Implementation Geschäftsprozesse und -modelle
  (bspw. Ablauf einer Bestellung, Modell eines Artikels)
- **Datenhaltung** Data / Persistence Layer
  Persistente Datenspeicherung, Datenlogik
  bspw. Speicherung der Artikeldaten in RDBMS

- Diese Aufteilung lohnt sich quasi immer, unabhängig von physischer Verteilung
  → SoC (Separation of Concerns)
  → SRP (Single Responsibility Principle)
  → **Modularisierung!**

Verfeinerung Präsentationsschicht

- Reine Präsentation (Formulare, Dialoge etc.) bei Rich-GUI/Client:
  noch stärker von Präsentationslogik & Daten trennen
- bspw. mit Model View Control (MVC):
  Modelle für Präsentation, wiederverwendbare Views, Präsentationslogik im Control
- Trennung zwingend notwendig bei Thin-Clients (bspw. Web/HTML):
  View durch HTML/CSS realisiert, Präsentationslogik und Modelle
  im Client (JS/JSON/XML) und im Server (Servlet, JSP, JSP/POJO's etc.)

Verfeinerung Geschäftslogikschicht

- Businessfunktionen → **Services** / **Businessmethoden**
  Enthält Klassen für (Geschäfts-)Prozesse, arbeitet mit Domain Model
  (bspw. `BestellService` [Kunde bestellt Artikel])
  Kann technologisch weiter verfeinert werden bspw. Business Service als Web Service anbieten
- Business Objects, Domain Objects →**Domain Model**
  Reines objektorientiertes Modell, vollständig unabhängig von Präsentation & Persistenz, enthält Daten und Methoden (bspw. `Artikel`, `Kunde`, `Adresse`)

Verfeinerung Datenhaltungsschicht

- Trennung / Abstraktion der reinen Datenlogik:
  unabhängig vom physischen Datenmodell & verwendetem (R)DBMS
- O/R-Mapping: Einsatz eines Persistenz-Frameworks
- Transparentes Einbinden von Legacy-Systemen, Abstraktion mehrerer Backend (DBMS-)Systeme

Abbildung 10: 6-Schichten-Architektur aus Verfeinerung der 3-Schicht-Architektur

7.4.3 1, 2, 3, oder n-Schichten?

Je mehr Schichten ein System enthält...

Positive Effekte:

- Bessere Strukturierung, einzelne Schichten kleiner und einfacher
  $\rightarrow$ besseres & schnelleres Verständnis
- Grössere Chance auf Wiederverwendung
- Höhere Flexibilität bspw. Austausch einzelner Schichten
- Tendenziell bessere Skalierbarkeit
- Einfachere & präzisere Planung / Schätzbarkeit
- Parallele & getrennte Entwicklung möglich

Negative Effekte:

- Komplexität des Systems wird grösser
- Mehr Schnittstellen $\rightarrow$ mehr Aufwand $\rightarrow$ mehr Planung

7.5 Sie sind sich der Bedeutung des Domänenmodells bewusst

- Vollständige Abstraktion der Geschäftsprozesse und -daten
  'Reines' objektorientiertes Modell der Domäne, vollständige Trennung der physischen Datenspeicherung, Interaktion „nur“ zwischen (Domain-)Objekten
- Logisches Beispiel: Klassen `Artikel` und `Lager`
  `Artikel`: Beschreibt konkreten Artikel, der einem Lagerort zugeordnet ist (entspricht Tupel)
  `Lager`: Beschreibt Lager, welches Artikel enthält, hat Fähigkeit Artikel zu reservieren, beziehen etc.
- Technische Beispiele:
  Enterprise Java Beans (Session & Message Driven),
  Domain Model mit POJOs, übergeordnete Business-Services

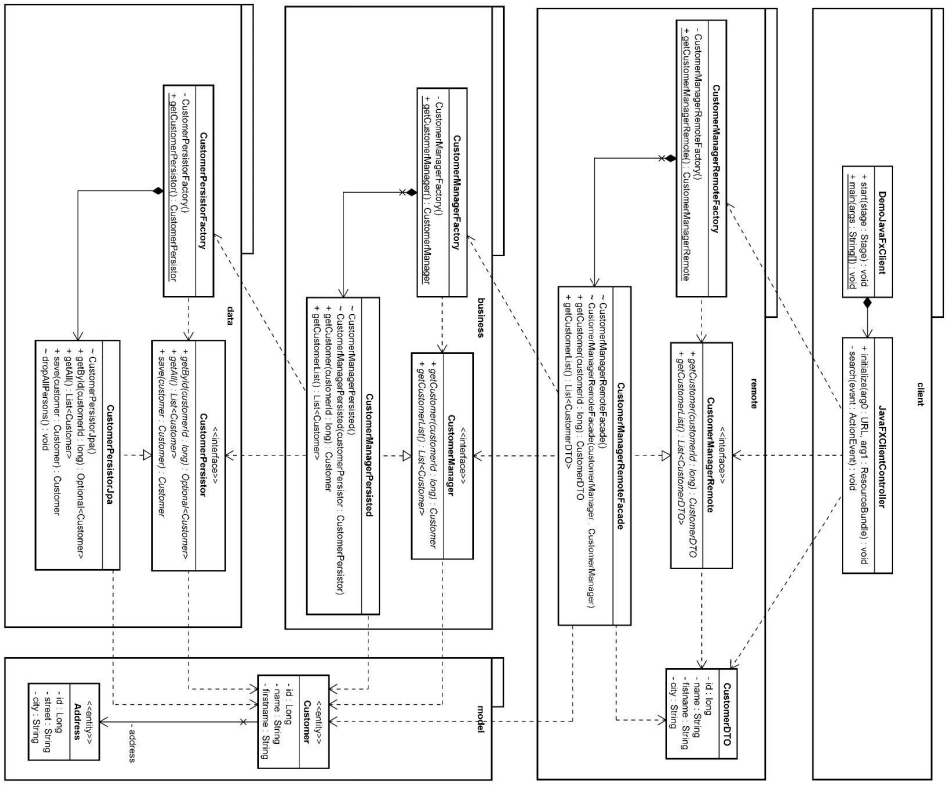


Abbildung 11: Einordnung des Domain Model in der Schichtenarchitektur

Domain Model - Diskussion Realisiert praktisch alle Vorteile einer logischen 3-Schicht-Architektur, besonders:

- Entkopplung von Präsentation und Datenspeicherung
- Hohe Wiederverwendung (des Modelles)
- Gute Wart- und Erweiterbarkeit
- Starke Strukturierung, leichteres Verständnis
- Skaliert: auch für grosse / komplexe Enterprise-Systeme geeignet

Nachteile:

- Abstraktion / Mapping des Persistenzmodells verursacht Overhead (bspw. durch OR-Mapping)
- Effizienter Datenaustausch zw. Schichten ist herausfordernd
  (wiederholtes Mapping zur Entkopplung)
- *Als Übung: Entwerfe ein Domänenmodell (fachliches Datenmodell) einer Anwendung, deren Funktion und Inhalt du gut kennst (gute Vorbereitung für Prüfungsaufgaben ;-))*

Schichtenarchitekturen - Bilanz

- **Es kommt auf die Grösse an!**
- „Im Kleinen“:
  Schichten funktionieren super und ergeben sich fast von selbst.
  Herausforderung: Ab wann ist Interface sinnvoll / nötig?, zu viele Schichten gibt es fast nicht, bspw. OSI-Schichtenmodell
- „Im Grossen“:
  Schnittstellen sind Pflicht, was sich bspw. aus Aggregation aus Modulen fast selbst ergibt. Man tendiert hier eher zur Minimierung (so viel wie möglich)
- **Problem:** Nicht Anzahl („Höhe“) der Schichten ist die Herausforderung, sondern zu erkennen, wenn eine Schicht (oder ganzer Stack) zu breit wird.
- Domäne einer App wird evtl. zu gross, als Konsequenz werden Schichten zu breit und es sind Dinge darin zusammengelegt, die nichts mehr miteinander zu tun haben.
  Weiteres Mittel zur Partitionierung wird benötigt $\rightarrow$ *Microservices?*

7.6 Sie haben eine Vorstellung was Microservices sind

> *“In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery.”*
> *- Martin Fowler*

- Eine Applikation wird in mehrere, kleine Services aufgeteilt
- Jeder Service läuft in eigenem Prozess / auf eigener Plattform
- Leichtgewichtige Kommunikation (meist RESTfull / HTTP / JSON)
- Voneinander unabhängig deploybar
- Automatisiertes Deployment (DevOps, PaaS, IaaS)

Aufteilung einer Applikation in kleine Services

- Applikation (primär vertikal) in mehrere, (möglichst) eigenständige Teile aufteilen
  $\rightarrow$ Teile sollen autark arbeiten können & auf eigene Daten(-modelle) zugreifen können
- Domänenmodell muss in verschiedene „bounded context“ aufgebrochen werden
  $\rightarrow$ Gut für Design, aber nicht einfach, Datenhaltung ist so auch getrennt!
- Einzelne Teile sollten nicht (direkt) miteinander kommunizieren, werden über das GUI oder vorgelagerten Gateway orchestriert
- Microservices sind unterschiedlich gross, meistens gilt:
  Applikation > Microservice $\geq$ Modul

Jeder Service hat eigenen Prozess / Plattform

- Services als eigenständige Prozesse
  - Unterschiedliche Plattformen (OS, Programmiersprach etc.), in virtualisiertem Container
  - Plattformen für ganze App. müssen nicht identisch sein, können schrittweise geändert werden
- Services laufen in echter Parallelität als verteiltes System
  - Ganzes Potenziell, alle Herausforderungen von verteilten Systemen:
    $\rightarrow$ Netzwerkkommunikation, Latenz, Skalierung, Ausfall etc.
  - Nutzt Client mehrere Services asynchron, erhöht sich Performance
- **Komplexität des Systems erhöht sich!**

Leichtgewichtige Kommunikation

- Populär: JSON-basierende REST-Schnittstellen (sind „einfach“)
- Kritisch hinterfragen: JSON/REST „leichtgewichtig“ im vergleich mit schnellen, effizienten binären Protokolle wie RMI, RPC etc.?
- Auf HTTP(S) basierende Kommunikation leicht zu implementieren & automatisiert testbar
  $\rightarrow$ Authentifizierung & Verschlüsselung abgedeckt, gute Akzeptanz im Operating, bestehende / bekannte Protokolle

Voneinander unabhängig deploybar / releasebar

- Microservices: eigenständige Projekte / Releases, erst durch gemeinsame „Orchestrierung“ eine Applikation $\rightarrow$ kleinere Einheiten flexibler entwickeln, Änderungen mit tieferem Risiko & Erweiterungen leicht ergänzbar (OCP)
- Unabhängig deploybar: Gesamte Applikation damit umgehen können, dass einzelne Teile/Services ausfallen und nicht verfügbar sind
- Häufigere Ausfälle $\rightarrow$ Resilienz wird wichtig
  (Bei einem Teilausfall nicht vollständig versagen)

Deployment automatisiert (DevOps)

- Automatisierung ist für Microservices ein Muss-Feature
  (mit schnelleren Start-/Stop-Sequenzen)
- Klassische, monolithische Deployments: (halb-)manuelle Deployments und Start-/Stop-Sequenz war häufig bei „schwergewichtigen“ Applikationen $\rightarrow$ Manuell, fehleranfällig, Dauer im Bereich weniger Minuten, mit Microservices würde sich diese Tätigkeit um die Anzahl Microservices verlängern

Klassisch vs. Microservices

- Schichten (horizontal) und Microservices (vertikal) sind orthogonal
- Am Schnittpunkt zwischen diesen befindet sich idealerweise ein Modul
- Entscheidend ist Grösse und Schnitt der Release- und Deployment-Einheiten
- Gegenseitige Abhängigkeiten sollen minimal sein
  - Möglichst wenig und gering, ideal keine (minimale Kopplung)
  - Microservices forcieren Modularisierung, verlagern das Problem ins Deployment (Operating)

Vorteile von Microservices:

- einzeln überschaubar, unabhängig, gut wartbar
- relativ schnell & flexibel (agil) anpassbar
- von kleinen, schlagkräftigen Teams entwickelbar
- unterschiedliche Plattformen, Sprachen & Technologien

Nachteile von Microservices:

- Hohe Anfoderungen an Operatin (DevOps)
- Höhere Komplexität durch Asynchronität & Resilienz
- Neue Herausforderung (cross-cutting concerns) wie
  Sicherheit, Überwachung, Monitoring & Logging

Herausforderung: Design (bounded contexts) / verteilte Applikationen

7.7 Sie kennen den Unterschied zwischen synchroner und asynchroner Kommunikation in verteilten Systemen

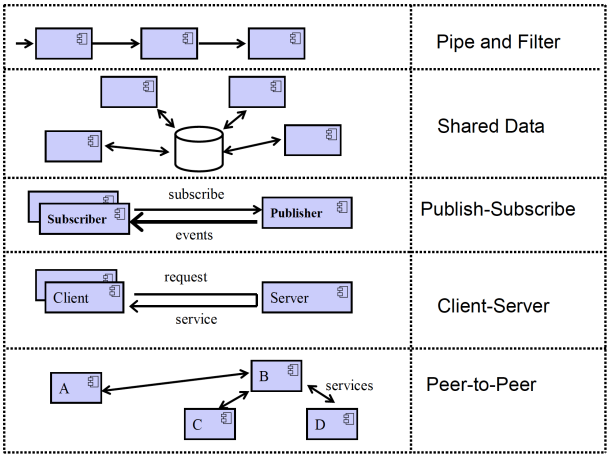
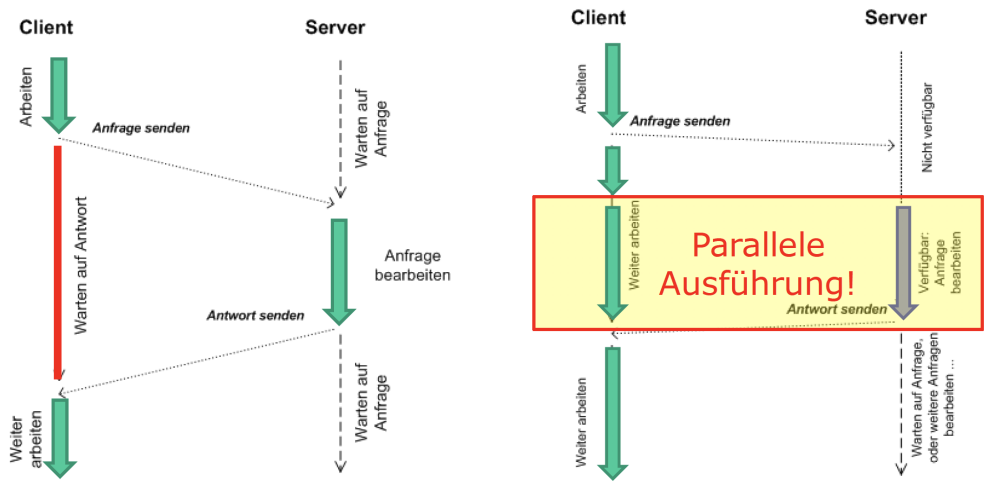


Abbildung 12: Muster für verteilte Applikationen

- Einzelne Teile (Teilsysteme, Komponenten, Module, Services, Schichten etc.)
  auf mehrere versch. Rechner (hosts, tiers) verteilt
- Konsequenzen: Teile laufen in einzelnen unabhängigen Prozessen & in „echter“ Parallelität
- Anforderungen:
  - Auswahl geeigneter Kommunikationstechnologie zw. Teilen
  - Einzelne Teile müssen sich finden & kenne
  - Aufwändigeres Deployment, mehr und häufiger, Abhängigkeiten beachten & koordinieren
  - Mit Teilausfällen muss umgegangen werden können

7.7.1 Kommunikationsmodelle



(a) Synchrone Kommunikation (b) Asynchrone Kommunikation

Abbildung 13: Kommunikationsmodelle (synchron & asynchron)

Synchrones Modell / Kommunikation

- Server muss verfügbar sein, bevor Kommunikation durch Client gestartet werden kann
- Sender muss auf Antwort des Empfängers warten, bevor er weiterarbeiten kann (blockierend)

Vorteile:

- Einfacher in Implementierung
- Zugriff auf gemeinsame Ressourcen weniger kritisch, da zeitlicher Ablauf überschaubar

Nachteile:

- „Wartezeit“ beim Client: negative Auswirkung auf Performance (verlorene Zeit?)
- Enge Kopplung zw. Sender / Empfänger, weniger robust bspw. bei Ausfall
- Server muss verfügbar sein & auf Anfragen warten

Asynchrones Modell / Kommunikation

- Client kann Nachricht senden, egal ob Server für Empfang bereit oder nicht
- Client wird nicht blockiert, kann nach Übermittlung der Anfrage weiter arbeiten
- Antwort trifft später asynchron ein oder wird abgeholt

Vorteile:

- Lose Kopplung Client / Server → robuster
- Keine gegenseitige Blockierung
  (Server muss nicht Anfrage erwarten, Client muss nicht auf Antwort warten & arbeitet weiter)

Nachteile:

- Komplexere Implementierung: zusätzliche Infrastruktur für Kommunikation (Queues)
- Wenn Client ohne Antwort nicht weiterarbeiten kann, macht async. Anfrage wenig Sinn

8 Grafisches User Interface - mit JavaFX

9 Persistierung - JPA und OR Mapping

10 Persistierung - Java Persistence Query Language (JPQL)

11 Kommunikation - Remote Method Invocation (RMI)

12 Web Services - REST